Response Letter for Submission
"Accelerating Directed Densest Subgraph Queries with Software and Hardware Approaches"

CHENHAO MA, YIXIANG FANG, REYNOLD CHENG, LAKS V.S. LAKSHMANAN, XIAOLIN HAN, and XIAODONG LI

Dear Editor, Associate Editor, and Reviewers,

Thank you very much for the invaluable comments. We have addressed all the issues raised by reviewers. Please find below our point-by-point response to the concerns raised, as well as the revision actions taken. Note that the figures and tables mentioned in this letter are indexed by Roman numerals (e.g., Fig. II). We have also highlighted the revised parts in our manuscript. Table/figure numbers in Arabic numerals refer to the paper.

Yours sincerely,
Chenhao, Yixiang, Reynold, Laks, Xiaolin, and Xiaodong

1 COMMENTS FROM REVIEWER #1

> *1. In Sec. 7.2.1, the authors mention Core-Approx is slightly faster than proposed CP-Approx. It would better if author can provide some explanations on why CP-Approx is slightly slower than Core-Approx.*

Thanks for your comments. Below please find our explanation.

(1) `CP-Approx` offers more flexibility to control the accuracy guarantee. `CP-Approx` is a $(1+\epsilon)$-approximation DDS algorithm, while `Core-Approx` is a 2-approximation DDS algorithm. Hence, to achieve flexibility, it is reasonable to pay for the cost.
(2) The complexity of the CP-Approx algorithm is $O(\sum_{c=\frac{1}{\sqrt{n}}}^{\sqrt{n}} 16(\sqrt{c}+\frac{1}{\sqrt{c}})m \max \sqrt{c}d^{+}\max, \frac{1}{\sqrt{c}}d^{-}\max)$, when $\epsilon = 1$, while the complexity of the Core-Approx algorithm is $O(\sqrt{m} \cdot (n+m))$. We can observe that the complexity of CP-Approx is higher than that of Core-Approx, as $m \cdot \max \sqrt{c}d^{+}\max, \frac{1}{\sqrt{c}}d^{-}\max$ is larger than $\sqrt{m} \cdot m$.
(3) Compared to other $(1+\epsilon)$-approximation DDS algorithms, `CP-Approx` is still much faster as shown in our experiments.

We have included the explanations in Sec. 7.2.1. in the revised version.

> *2. In Sec. 6, the authors apply 3 strategies on vertex assignment on GPU. There is a more common way of assignment which is assigning one vertex (i.e., all its adjacency list items) to a warp. I am curious to see explanations on why adaptive strategy is better than warp-level assignment.*

Thanks for your suggestion. We have tested the warp strategy. The results (including the warp, adaptive, block, and thread strategies) are reported in Figure I. We found that the adaptive strategy generally outperformed the other two strategies. The warp strategy showed better performance than the block strategy for large block sizes, which can reduce

Authors' address: Chenhao Ma; Yixiang Fang; Reynold Cheng; Laks V.S. Lakshmanan; Xiaolin Han; Xiaodong Li.

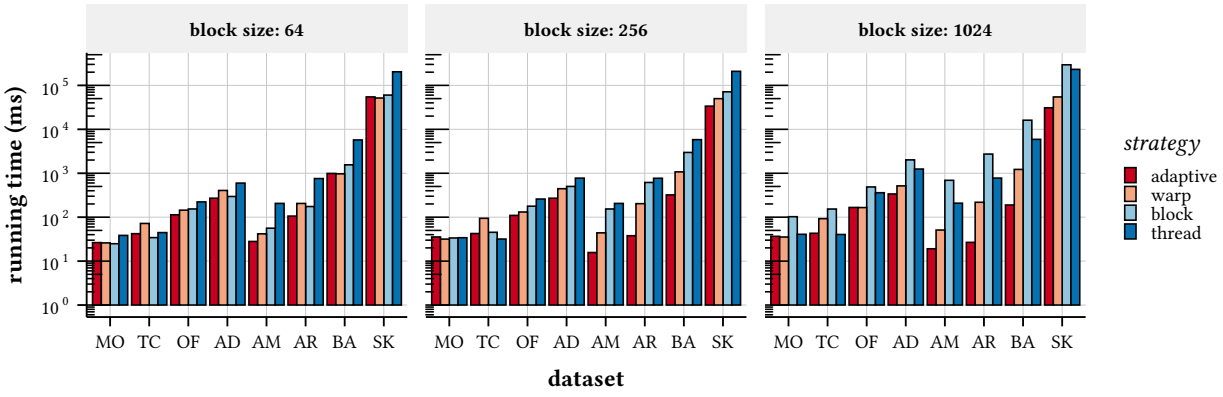

Fig. I. Running time of Frank-Wolfe iterations under different strategies.

computing resource waste. Nevertheless, the warp strategy still suffered from the imbalance among vertex degrees, as the warp size is 32 and there can be many vertices with degrees much less than 32. Our findings suggest that the adaptive strategy is more effective in utilizing the resources of the GPU, leading to higher performance.

We have also included the warp strategy in Sec. 6 and 7 in the revised paper.

> *3. It is necessary to report what type of GPU you were using in experiments.*

The GPU we used in the experiments is GeForce GTX 2080 Ti 11GB GPU.

2 COMMENTS FROM REVIEWER #2

> *Weak points.*
>
> *W1. Few details on the GPU acceleration part are missing*
>
> *Detailed comments.*
>
> *D1. Can you please report the total end-to-end speed up when considering CPU-GPU data movements? I see that you are close to the theoretical maximum for the SK graph, but does this consider CPU-GPU data movements? How about the other graphs? If you are getting almost 4x for SK, I would expect that for MO TC OF datasets GPU execution might be not worthy.*
>
> *D2. It would be useful to also know which kind of GPU are you using and which generation of PCIe, as well as the total used GPU memory.*

Thank you for the suggestion.

(1) **D1.** We report the data movement time, overall running time with GPU enabled, and the end-to-end speedup. Here, the adaptive strategy is adopted and the block size is set to 1024. We can see that the speedup is less than 1 on some cases. The reason is that the running time on those datasets (e.g., MO, TC, OF) is less than 0.1

seconds, while the GPU initialization can incur up to 3 seconds of latency[1]. Due to this issue, the GPU speedup is designed for the large-scale datasets, e.g., SK and BA.

(2) **D2.** The GPU we used is GeForce GTX 2080 Ti 11GB GPU. The PCIe generation is 3.0. The total used memory over different datasets is reported in Table II. We can observe that the edges involved in the Frank-Wolfe computation can be well fit in GPU memory, as we have applied Theorem 5.1 to reduce the number of edges to be used in the Frank-Wolfe iterations.

Table I. Breakdown of total time (in seconds) for GPU-enabled version and end-to-end speedup.

Dataset	Data movement	GPU comp.	Pre/post process	GPU ver. Total	CPU ver. Total	Speedup
MO	0.0081	0.0367	3.337	3.3818	0.1692	0.05
TC	0.0079	0.043	3.2798	3.3307	0.0856	0.03
OF	0.0374	0.1664	3.5683	3.7721	3.2753	0.87
AD	0.1209	0.3378	3.5789	4.0376	13.584	3.36
AM	0.0228	0.0191	5.8581	5.9	3.6589	0.62
AR	0.0881	0.0267	13.0513	13.1661	22.7376	1.73
BA	0.9461	0.1894	118.0355	119.171	214.3584	1.80
SK	118.765	30.9194	2390.4377	2540.1221	9851.7132	3.88

Table II. Maximum GPU memory used over different datasets.

Dataset	Memory (in MB)	Data transferred between CPU and GPU (in MB)
MO	207	1.93
TC	207	2.01
OF	207	35.83
AD	207	177.08
AM	213	6.06
AR	257	29.72
BA	257	241.33
SK	5,585	44,661.69

> *Minor comments:*
>
> *M1. "strategies, We" -> we*
>
> *M2. "assigning each of them to a block is extravagant" -> ?*

Thanks. We have fixed the typos and grammar errors.

[1] https://deci.ai/blog/measure-inference-time-deep-neural-networks/

3 COMMENTS FROM REVIEWER #3

> * *The discussion and evaluation of the GPU accelerations are quite incomplete. Please include the following.*
>
> *(1) Description of the actual GPU used, clock frequency, and amount of GPU device memory*
>
> *(2) Size of the data transferred (GB) between CPU and GPU for each workload*
>
> *(3) Breakdown of total time into the following components: setup & data preparation (if any), data movement, GPU computation time for each workload*
>
> *(4) Is the data fully resident in GPU memory for each workload?*
>
> *(5) How to handle situations where the graph does not fit into GPU device memory?*
>
> *(6) What was the memory bandwidth utilization for the different workloads?*
>
> *(7) What was the observed branch efficiency/warp divergence for the different workloads?*
>
> *(8) What was the observed SM efficiency for the different workloads?*

(1) The GPU we used is GeForce GTX 2080 Ti 11GB GPU, and the base clock frequency is 1350 MHz. The PCIe generation is 3.0.

(2) When using the adaptive strategy, the maximum GPU memory used and the size of the data transferred between CPU and GPU for each dataset are reported in Table II. For other strategies, the memory usage is quite similar, so we omit them. The total size of the data transferred can be larger than the maximum GPU memory used, because there can be multiple Frank-Wolfe calls and multiple rounds of data transfer.

(3) Table I shows the breakdown of total time into data movement, GPU computation time, and preparation and postprocess time. We also show the end-to-end speedup of the GPU-enabled computation compared to the CPU version. We can observe from the table that the GPU-enabled speedup is more suitable for large-scale graphs (around billion-scale), as the GPU initialization brings an overhead of 3 seconds.

(4) Yes, the data used to compute Frank-Wolfe iterations is fully resident in GPU memory, which can also be found from Table II.

(5) In our algorithm, the whole graph data does not need to be fit into GPU device memory. Only the edges related to the Frank-Wolfe computation are needed. The number of edges needed by Frank-Wolfe can be reduced by Theorem 5.1 with $[x, y]$-cores. From Table II, we can also find that even on the billion-scale graph, SK, the maximum GPU memory usage is only around 5GB, while the capacity is 11GB. For the cases that the related edges can not be fit into GPU memory, though ever rare, we will consider to use unified memory to also make use of CPU memory.

(6) We performed memory workload analysis using Nsight Compute[2], comparing different GPU parallel computing strategies with the block size set to 1024. Table III shows the results, reporting two metrics: *Max Bandwidth*, which shows the percentage of the peak memory bandwidth utilized, and *Mem Busy*, which represents the percentage of time the memory subsystem was busy. In *Max Bandwidth*, the value "inf" means that the application was able to fully saturate the available memory bandwidth. We observed that both memory utilization metrics increased with dataset size. In addition, the block strategy consistently resulted in higher memory usage compared to the adaptive strategy.

[2]https://developer.nvidia.com/nsight-compute

(7) and (8) We used Nsight Compute to perform compute workload analysis, comparing different GPU parallel computing strategies with a block size of 1024. Table IV shows the results, reporting three metrics: *SM Busy*, which represents the percentage of time that the Streaming Multiprocessors (SMs) were executing instructions during the kernel execution, *Avg. Active Threads Per Warp*, which shows the average number of threads in a warp that were active, and *Achieved Active Warps Per SM*, which denotes the average number of active warps per SM during kernel execution.

We observe that the adaptive strategy and block strategy have very little warp divergence, as the average number of active threads per warp is close to 32. The adaptive strategy achieved better performance than the block strategy, as it achieved a higher number of active warps per SM. This suggests that the adaptive strategy is better able to utilize the resources of the GPU, leading to higher performance.

Table III. Memory Analysis Results

Dataset	Max Bandwidth (in %)				Mem Busy (in %)			
	Adaptive	Block	Thread	Warp	Adaptive	Block	Thread	Warp
MO	0.85	2.59	0.36	inf	0.92	11.91	0.43	1.05
TC	0.49	2.86	0.23	inf	0.59	13.12	0.31	1.39
OF	0.85	2.59	0.36	inf	0.59	13.12	0.31	5.22
AD	7.65	2.97	1.62	inf	7.8	13.68	1.62	6.56
AM	0.2	2.29	0.07	inf	0.25	10.48	0.09	0.51
AR	9.66	3.05	0.21	inf	11.06	13.95	0.23	4.77
BA	6.1	2.88	inf	inf	6.36	13.32	0.85	1.99
SK	20.45	21.2	inf	inf	31.51	35.63	19.8	72.99

Table IV. Compute Workload Analysis

Dataset	SM Busy (%)				Avg. Active Threads Per Warp				Achieved Active Warps Per SM			
	adaptive	block	thread	warp	adaptive	block	thread	warp	adaptive	block	thread	warp
MO	17.12	18.13	14.61	18.41	30.87	31.84	19.79	25.64	37.71	31.73	13.33	34.63
TC	15.31	18.36	12.83	19.03	30.85	31.81	22.49	23.93	38.92	31.17	17.37	34.38
OF	15.32	18.39	12.95	17.78	31.19	31.85	20.27	26.09	36.56	31.02	13.78	33.61
AD	15.93	18.4	10.79	17.47	31.07	31.86	18.62	26.48	36.47	30.96	16.83	33.62
AM	15.56	17.89	7.52	16.69	31.48	31.85	10.65	24.97	39.66	33.80	3.79	35.82
AR	13.13	18.83	7.42	21.41	31.13	31.77	18.44	22.76	35.24	30.21	9.92	29.83
BA	15.43	18.3	7.02	16.66	31.11	31.9	23.75	27.81	35.57	31.63	8.87	35.75
SK	6.96	6.85	6.63	2.56	31.89	31.9	31.92	31.46	30.91	31.04	30.68	31.33
Average	14.345	16.89375	9.97125	16.25125	31.19875	31.8475	20.74125	26.1425	36.38	31.445	14.32125	33.62125

> * *Please back up/substantiate the illustration in Fig 6.1 with metrics that you actually measured on the GPU you used. Currently there is no empirical evidence shown in support of the illustrations.*

The illustration in Fig 6.1 can be substantiated by statistics in Table IV. We can make the following observations.

(1) the number of average active threads per warp under the thread strategy is much lower than other strategies, which may indicate the warp divergence rate is high for the thread strategy.
(2) The warp divergence rates of the adaptive strategy and block strategy are lower as the average numbers of active threads are close to 32.
(3) The adaptive strategy outperforms the block strategy with a higher number of achieved active warps per SM, which may indicate the adaptive strategy can further reduce the number of idle threads illustrated in Fig 6.1(b).
(4) The warp strategy assigns each vertex to a warp. Hence, each block can have several vertices assigned to different warps. As different vertices can have different degrees, the warps working on the vertices with higher degrees may take longer to complete their computation, causing some imbalance and leading to a lower average number of active threads per warp.

> * *Are the CPU implementations all single-threaded? if not, please mention the degree of parallelism used. Otherwise, please evaluate parallel implementations of the algorithms on the CPU and report on performance scalability with #cores. Modern servers can have many tens of cores per CPU socket, in addition to having multi-socket capability. Speedups for 5 of 8 workloads in Table 7.6 may be achievable on modern servers, if the baseline is a single-core CPU.*

The original CPU implementations were all single-threaded. We have implemented the multi-thread CPU version by using OpenMP, the performance is reported in Table V. However, we found that our GPU version outperforms the 32-thread CPU version, particularly on large scale datasets.

Table V. Running time of Frank-Wolfe with single-thread CPU, multi-thread CPU, and GPU with adaptive strategy.

Dataset	Frank-Wolfe (1 thread)	Frank-Wofle (32 thread)	Frank-Wolfe (GPU)
MO	0.1538	0.1469	0.0367
TC	0.0781	0.1155	0.043
OF	3.122	0.9573	0.1664
AD	13.2481	3.3340	0.3378
AM	0.9604	0.3585	0.0191
AR	12.1638	3.1542	0.0267
BA	97.8741	28.8372	0.1894
SK	7150.3231	1464.98	30.9194

> * *The manuscript does not seem to be self-contained. For example, section 4.1 presents an LP formulation inspired by that in [10], and the differences mentioned, but the formulation from [10] is not described first. This makes it difficult to read. Some more insight/explanation of the formulation would be appreciated.*

The LP formulation in [10] is formulated as following.

$$
\begin{aligned}
\mathsf{LP}(c) \qquad \max \quad & x_{\text{sum}} = \sum_{(u,v)\in E} x_{u,v} & \\
\text{s.t.} \quad & x_{u,v} \geq 0, & \forall (u,v) \in E \\
& x_{u,v} \leq s_u, & \forall (u,v) \in E \\
& x_{u,v} \leq t_v, & \forall (u,v) \in E \\
& \sum_{u\in V} s_u = \sqrt{c}, & \\
& \sum_{v\in V} t_v = \frac{1}{\sqrt{c}}. &
\end{aligned}
$$

The LP formulation tries to maximize the total weight of the edges subject to multiple constraints. The constraints include: (a) non-negativity, whereby all variables must be non-negative; (b) capacity constraints, which dictate that the weight of an edge (u, v) cannot exceed the minimum of the capacities of its endpoints, i.e., $x_{u,v} \leq s_u$ and $x_{u,v} \leq t_v$ (c) capacity sum constraints, which require that the sum of the capacities of the vertices must be fixed at a value related to the ratio of the sizes of two sets. The variables s_u, t_v, and $x_{u,v}$ indicate the inclusion of a vertex u/vertex v/edge (u, v) in an optimal densest subgraph according to whether the variable value is larger than 0, when $c = \frac{|S^*|}{|T^*|}$.

Our LP formulation is presented below. Our LP relaxation is similar to the LP relaxation in [10], but they are different since we have an additional constraint $a + b = 2$. When $a = 1$ and $b = 1$, our LP formulation is exactly the same as the one in [10]. The additional constraint allows us to establish the connection between the optimal value of the $\mathsf{LP}(c)$ for a fixed c, denoted by $\mathsf{OPT}(\mathsf{LP}(c))$, and the density of the DDS, and the connection will play a key role in reducing the number of LPs examined.

$$
\begin{aligned}
\mathsf{LP}(c) \qquad \max \quad & x_{\text{sum}} = \sum_{(u,v)\in E} x_{u,v} & \\
\text{s.t.} \quad & x_{u,v} \geq 0, & \forall (u,v) \in E \\
& x_{u,v} \leq s_u, & \forall (u,v) \in E \\
& x_{u,v} \leq t_v, & \forall (u,v) \in E \\
& \sum_{u\in V} s_u = a\sqrt{c}, & \\
& \sum_{v\in V} t_v = \frac{b}{\sqrt{c}}, & \\
& a + b = 2. &
\end{aligned}
$$

We have added more explanation of the formulation in Section 4.1 of the revised paper.

> ** It seems from Fig 7.2 that Core-Approx achieves low approximation ratios on these workloads, although the worst-case is 2. What is its average approximation ratio and how does it compare to CP-Approx? I understand that CP-Approx takes on average 2.68x the time taken by Core-Approx, but on average is it at least 2.68x better? From a cost-benefit perspective, does CP-Approx have an advantage over Core-Approx?*

The average empirical approximation ratio of Core-Approx over the eight datasets is 1.075, while the average empirical approximation ratio of CP-Approx when $\epsilon = 1$ is 1.013. Hence, the approximation ratio provided by CP-Approx is around 5% lower than Core-Approx.

Apart from providing lower empirical approximation ratios, CP-Approx also offers the flexibility on the approximation guarantee since ϵ can be any positive real value. It can be quite helpful to explore the approximate DDS's with different values ϵ as shown in Section 7.2.4.

> * *What approximation ratio could be achieved by Core-Approx for the results in Fig 7.3? You could fix $\epsilon = 2$ for it and show a flat line for comparison.*
>
> * *Please add similar analysis for the other 6 workloads in Fig 7.3. Why were only TC and BA chosen for fig 7.3?*

Thanks for your suggestion. We have included the result of Core-Approx and the plots for the six other datasets. The running time and density results can be found in Figure II and Figure III, respectively. The full result can not be fit into the main paper due to the page limit. We have put these extra experimental results in the GitHub repo, and refer the readers to the repo for more details. We can observe that the running time of CP-Approx, VW-Approx, and Flow-Approx decrease along with the increase of ϵ. Generally, the density also decreases along with the increase of ϵ. But, there may be some fluctuates as ϵ will also affect which values of $\frac{|S|}{|T|}$ to be enumerated. As the total number of values of $\frac{|S|}{|T|}$ to be examined within $[\frac{1}{n}, n]$ is $O(\log_{1+\epsilon} n)$, the chosen values for different ϵ are different. Different values of $\frac{|S|}{|T|}$ may result in different densities of the final returned subgraphs.

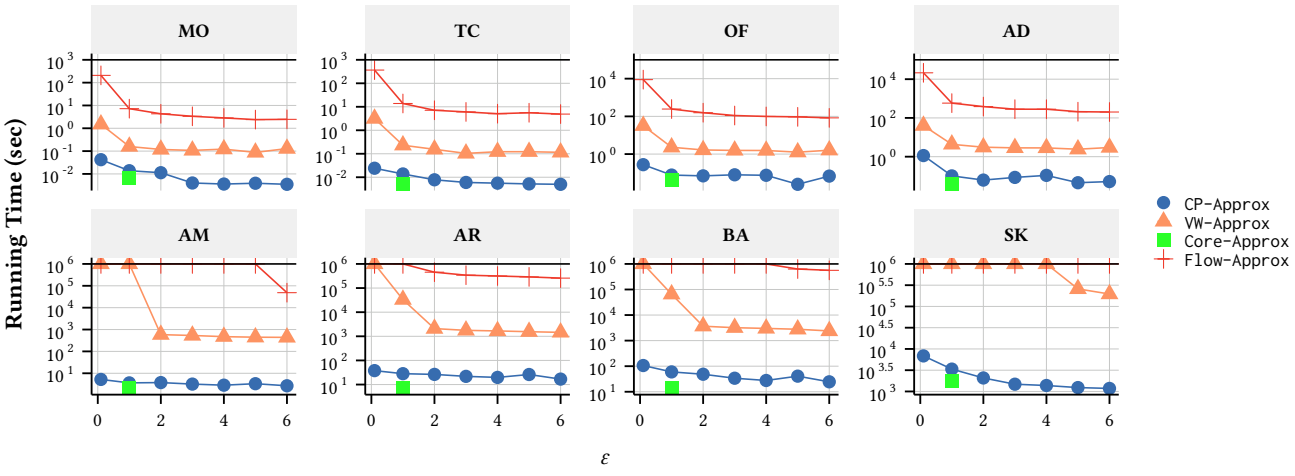

Fig. II. Effect of ϵ w.r.t. running time.

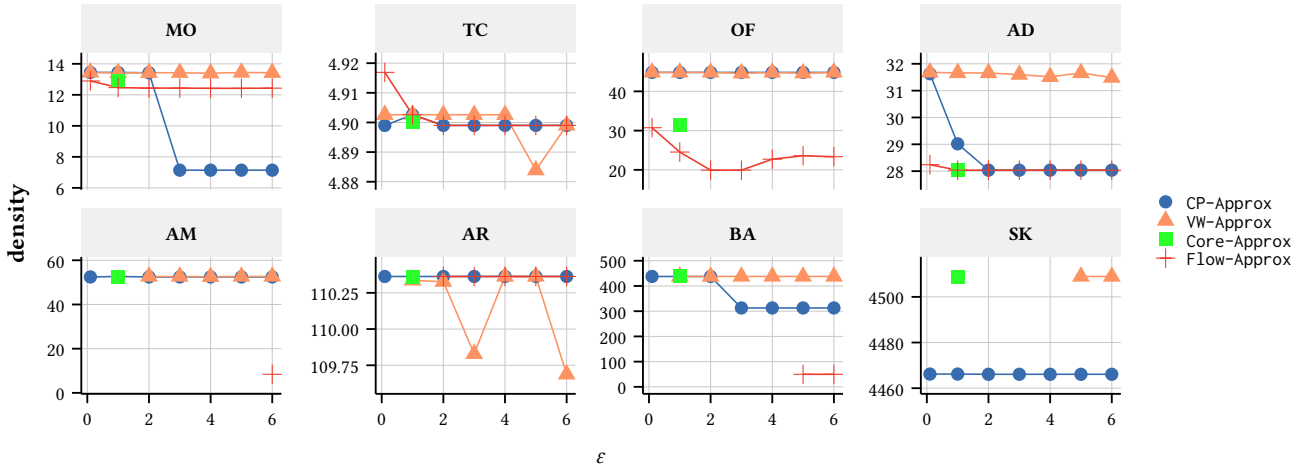

Fig. III. Effect of ϵ w.r.t. the density of the returned subgraph.

> * *In Fig 7.2, why couldn't CP-Approx find the solutions that Core-Approx found for MO, OF, AD? CP-Approx found more accurate solutions (that were not necessary for this value of ϵ) and ran slower. This seems that it is failing to take advantage of the flexibility that a user may be ready to provide in terms of relaxed approximation ratios.*
>
> * *In Fig 7.2, why could neither CP-Approx nor Core-Approx find the solution found by Flow-Approx for OF? Could they have terminated faster if they had found that solution?*
>
> * *In Fig 7.3, density found by CP-Approx flattens out after $\epsilon = 2$ (TC) and 3 (BA). Does this mean that solutions with larger approximations (that could perhaps be found faster) do not exist for these graphs, or the algorithm failed to find them?*
>
> * *In general, I am unable to figure out how much fast one could go if one could fully utilize the gap to exact solution that the worst-case approximation ratio allows. I suspect there could be valid solutions further from optimal that may be found faster. Please discuss why this may/may not be so.*

Thanks for your great question. We believe these four questions can be discussed together.

According to Figure III, we can find that on some cases the subgraphs returned by `CP-Approx` have the smallest density among algorithms. Sometimes, the subgraphs returned by `VW-Approx` or `Flow-Approx` also have the smallest density. This is because the three algorithms used different strategies to find the approximate densest subgraph falling into the tolerated range of density. `CP-Approx`and `VW-Approx` uses the linear/convex programming based strategy, but `CP-Approx` advances on several points: less values of $\frac{|S|}{|T|}$ to be examined; tighter error estimation in the inner loop; smaller size of the graph to be processed. `Flow-Approx` uses a fixed number of blocking flow iterations to find the solution.

Further, we would like to argue that it is also hard to find the subgraph which has the smallest density and satisfies the error tolerance, which can be treated as another optimization problem. As three algorithms all find the subgraph with smallest density on some cases, we reckon it is not very appropriate to examine whether an algorithm makes

good use of the gap just by the density of the returned subgraph. Actually, our algorithm runs fastest in the three $(1+\epsilon)$-approximation algorithms, and the running time drops along with the increase of ϵ. It is reasonable to believe that our algorithm makes good use of the allowed gap.

Yes, there could be valid solutions further from optimal that may be found faster, at least for our algorithm. By default, we use $N = 100$ iterations for a batch of Frank-Wolfe computation and check whether there are some graphs satisfying the requirement. We find that if we reduce $N = 100$ to some smaller numbers, we can have a slightly shorter running time and a slighter lower density value on some cases. Besides, if N is set too small, this may also further increase the running time (as we may need to perform too many checks before we fulfill the requirement).

Table VI. Running time and density w.r.t. different N.

Dataset	N	100	50	10	5
AD	time	0.1368	0.1288	0.0851	0.1049
	density	29.0183	29.0183	28.2627	28.2627
OF	time	0.1434	0.0987	0.071	0.0677
	density	44.829	44.8205	44.7277	44.7107
AR	time	32.1289	28.922	26.9226	26.9718
	density	110.363	110.363	110.363	110.363
BA	time	47.0355	45.3611	39.8072	36.9742
	density	437.6348	437.6348	437.6348	437.6348

> ** Table 7.2 provides a useful perspective, but why was only the AD workload chosen? Please include the analysis for the remaining 7 workloads as well.*

Thanks for your suggestion. We have included the statistics about the DDS's w.r.t. different ϵ values for the remaining 7 workloads are reported in Table VII. We can find that the similar phenomena can also be observed on several other datasets, e.g., AD, MO, TC, and OF. For these four datasets, the subgraphs over different ϵ share close density values but can be quite different in terms of the vertices contained. For others, their returned subgraphs are relatively close in terms of density and similarity.

> ** Please describe how the graphs are represented in CPU and GPU memory. What is the memory size (GB) needed for the different workloads? What was the resident set size (RSS, GB) when running the different algorithms?*

The graphs are represented as the adjacency list in CPU memory. Specifically, for each vertex, we have a vector (in C++) to store the indexes of the edges containing the vertex. We also have an array to keep all the edges.

We report the maximum memory usage of all algorithms in Figure IV. The memory usage of `Flow-Exact` and `Core-Exact` are omitted because the results are very similar to that of `DC-Exact`. We observe that the memory costs of all algorithms are around the same scale because all algorithms take linear memory usage w.r.t. the graph size. Among those algorithms, `LP-Exact` needs more memory than others because we implemented `LP-Exact` via Google OR-Tools, which materializes all constraints in the LPs.

Table VII. Statistics of DDS's w.r.t. different ϵ values

Dataset	ϵ	Density	$\|S\|$	$\|T\|$	Similarity
AD	0	31.6811	453	195	1.00
AD	0.1	31.6299	443	197	0.98
AD	1	29.0183	913	2	0.43
AD	2	28.0357	1	786	0.16
MO	0	13.4471	155	143	1.00
MO	0.1	13.4471	158	153	0.98
MO	1	13.4423	1	28	0.12
MO	2	13.4240	151	147	0.98
TC	0	4.9169	4	73	1.00
TC	0.1	4.9026	5	90	0.89
TC	1	4.9026	5	90	0.89
TC	2	4.8990	1	24	0.49
OF	0	44.8517	189	187	1.00
OF	0.1	44.8290	193	193	0.98
OF	1	44.8290	193	193	0.98
OF	2	44.8290	193	193	0.98
AM	0	52.6758	4944	2	1.00
AM	0.1	52.6758	4944	2	1.00
AM	1	52.6758	4944	2	1.00
AM	2	52.4500	2751	1	0.71
AR	0	110.3630	1	12180	1.00
AR	0.1	110.3630	1	12180	1.00
AR	1	110.3630	1	12180	1.00
AR	2	110.3630	1	12180	1.00
BA	0	437.6965	99040	2	1.00
BA	0.1	437.6346	95762	2	0.98
BA	1	437.6346	95968	2	0.98
BA	2	437.6346	95968	2	0.98
SK	0	4509.6160	4510	4515	1.00
SK	0.1	4466.1966	8932	894	0.67
SK	1	4466.1966	8932	8942	0.67
SK	2	4466.1966	8932	8942	0.67

For our GPU-enabled algorithms, the maximum GPU memory used can be found in Table II. The maximum GPU memory usage under different strategies are quite similar. Hence, we only report the GPU memory usage under the adaptive strategy.

We have also updated Section 7 in the revised paper with the above statistics.

> ** Minor: - Sec 3.2: GPU memory is usually "high bandwidth memory" instead of "ultra-high bandwidth". - page 7: typo "in next section".*

Thank you. We have fixed the typos.

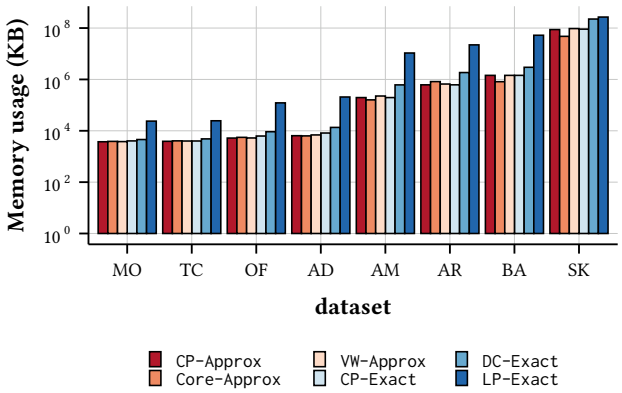

Fig. IV. Memory usage of algorithms.